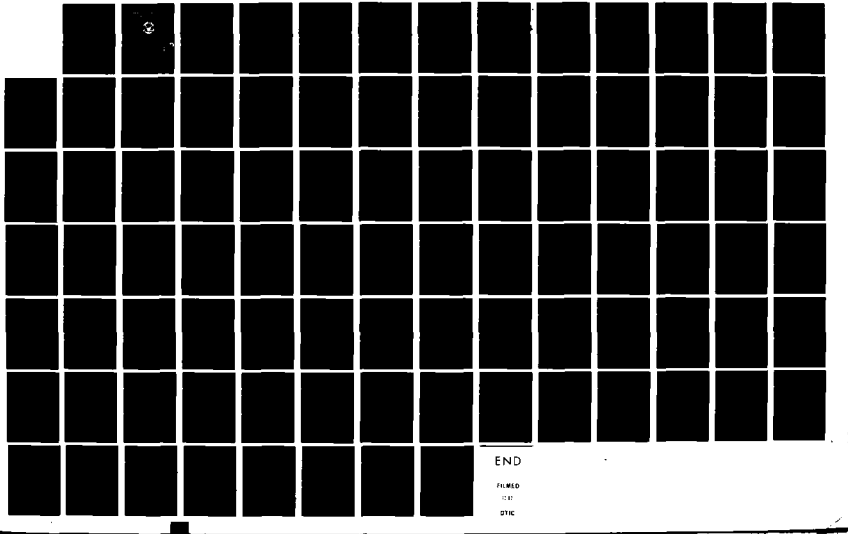
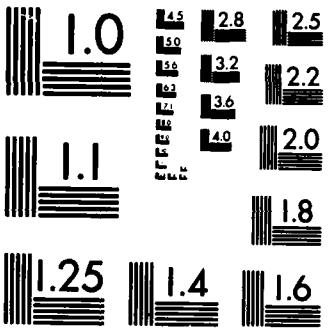
AD-A120 400 DISTRIBUTED DATABASE MANAGEMENT SYSTEM RECOVERY FROM NETWORK PARTITIONING(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA E E BRESANI JUN 82 1/1

UNCLASSIFIED F/G 9/2 NL





NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DISTRIBUTED DATABASE MANAGEMENT SYSTEM
RECOVERY FROM NETWORK PARTITIONING

by

Eduardo E. Bresani

June 1982

Thesis Advisor: Norman Lyons

Approved for public release; distribution unlimited

TO WHOM IT MAY CONCERN:

This package contains copies of official documents of the Naval Postgraduate School which are being transmitted for deposit with the Defense Technical Information Center, Alexandria, Virginia.

Should this shipment become lost in the mail or misrouted to an address other than the Defense Technical Information Center, it is requested that the shipment be redirected by using the postage paid mailing label, attached below.

Thank you,

Dudley Knox Library (Code 1420)
Naval Postgraduate School
Monterey, California 93940

Telephone: (408) 646-2341
Autovon 878-2341

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A120400	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Distributed Database Management System Recovery from Network Partitioning		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis June 1982
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Eduardo E. Bresani		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE June 1982
		13. NUMBER OF PAGES 88
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Distributed systems, database systems, networks, recovery, network partition, multiple operating partitions

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The purpose of this thesis is to analyze the operation of a distributed database management system under network partitions, review a number of existing methods proposed to deal with this problem and to present an alternate approach that will allow multiple operating partitions upon network partitioning.

When a network that supports a distributed database with redundant data becomes partitioned, each partition may

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

function separately. Due to this, independent updates at each partition may cause inconsistencies to arise. At network reconnection time such divergent data, in particular copies of the same data in different partitions have to be reconciled. There is no known general method for doing so. Existing solutions are often unacceptable because system availability is reduced. Two recently proposed methods that allow continuous operation of multiple partitions may work for certain applications but are not general enough.

Approved for public release, distribution unlimited.

Distributed Database Management System Recovery from Network Partitioning

by

Eduardo E. Bresani
Lieutenant, Peruvian Navy
B.S., Peruvian Naval Academy, 1975

Submitted in partial fulfillment of the
requirements for the degrees of

MASTER OF SCIENCE IN COMPUTER SCIENCE
and
MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

NAVAL POSTGRADUATE SCHOOL
June 1982

Author _____

Approved by: _____
Thesis Advisor

_____
Co-Advisor

_____
Chairman, Department of Computer Science

_____
Chairman, Department of Administrative Sciences

_____
Dean of Information and Policy Sciences

ABSTRACT

The purpose of this thesis is to analyze the operation of a distributed database management system under network partitions, review a number of existing methods proposed to deal with this problem and to present an alternate approach that will allow multiple operating partitions upon network partitioning.

When a network that supports a distributed database with redundant data becomes partitioned, each partition may function separately. Due to this, independent updates at each partition may cause inconsistencies to arise. At network reconnection time such divergent data, in particular copies of the same data in different partitions have to be reconciled. There is no known general method for doing so. Existing solutions are often unacceptable because system availability is reduced. Two recently proposed methods that allow continuous operation of multiple partitions may work for certain applications but are not general enough.

TABLE OF CONTENTS

I. INTRODUCTION 8

II. THE NETWORK PARTITION PROBLEM 12

A. INTRODUCTION 12

B. BASIC DEFINITIONS AND CONCEPTS 13

C. PROBLEMS AND ISSUES 16

1. Alternatives for System Operation Under Network Partitions 17

2. Correct Operation Under Network Partitions 20

III. PREVIOUS WORK ON PARTITIONING 25

A. INTRODUCTION 25

B. APPROACHES INVOLVING ONE OPERATING PARTITION . 26

1. Voting 26

2. Tokens 27

3. Primary Sites 28

4. Reliable Networks 29

C. APPROACHES INVOLVING MULTIPLE OPERATING PARTITIONS 30

1. Version Vector Mechanism 30

2. Semantic Knowledge 36

IV. ALTERNATE APPROACH TO AUTOMATIC CONFLICT DETECTION AND DATABASE RECONCILIATON 45

A. INTRODUCTION 45

B. DESCRIPTION OF THE APPROACH 46
1. Preliminary Definitions and Assumptions . 46
2. Conflict Detection and DataBase Reconciliation 50
3. An Example 62
C. EXTENSIONS TO THE APPROACH 67
1. Normal operation During Partition Merge . 68
2. Partial Partition Merges 71
3. Allowing irrecoverable external actions . 77

V. CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH . 80

LIST OF REFERENCES 84

INITIAL DISTRIBUTION LIST 87

LIST OF FIGURES

2.1. Restoring Mutual Consistency 22

3.1. Semantic classes and histories 40

4.1. Initial precedence graph 63

4.2. First modification to precedence graph 64

4.3. Second modification to the precedence graph . . 65

4.4. Third modification to the precedence graph . . . 66

4.5. Fourth modification to the precedence graph . . 66

4.6. Partial merges in a partition graph 73

4.7. Symmetric partial merges in a partition graph . 75

I. INTRODUCTION

In the past decade there has been considerable work, done on multiprocessor systems and computer networks. As consequence of this work the concept of distributed computing systems was developed and is presently a focus of intensive research in academia and Industry.

In particular Distributed Data Base Systems (DDBS) have became one of the more important research topics since many distributed systems are now being developed to provide users with convenient access to data via some kind of communications network.

A distributed database system has the potential advantages of greater data availability and reliability since data-items may be replicated and accessed at several sites throughout the system. We use the term "potential" because availability should increase with the number of copies of the data. If the multiple copies of data were read-only then availability will, in fact, be increased, however, when updates are also allowed, multiple copies may provide no improvement if mutual consistency among copies of the data is emphasized.

Mutual consistency requires, that if all update activity were to cease, then after some period of time all copies of

the same data will converge to the same value. There have been many algorithms published for maintaining mutual consistency during normal operation of a distributed database [1], [5], [19], [18] . Unfortunately, these algorithms do not consider mutual consistency in the face of network partitioning.

A network partition occurs when two or more disjoint subsets of sites in the network cannot exchange messages through the network (i.e. cannot communicate with each other) even though some or all of them are up and operational. A special case of network partitioning occurs if the only path between two or more sites is the communications network. In this case a single site crash cannot be distinguished from a network partition that separates that site from the rest of the network.

Network partitioning can completely destroy mutual consistency in the worst case and so the usual solution to deal with this problem has been to restrict operation during network partition in such a way that only one group of sites (i.e. within one partition) is allowed to do the updates. The basic idea behind this approach is that no update scheme is effective against partitioning in guaranteeing mutual consistency of data, unless data is always kept accessible only in one partition [19], [18] . The methods proposed vary in the way in which they select the set of sites allowed to do the updates.

However, these kind of schemes have as a major drawback that it may be unacceptable for the non-selected sites to shutdown operations while the network is partitioned. We must note that it is worthwhile to have all partitions in operation if (1) availability is just as important as consistency and (2) "conflicts" among copies of data can always be succesfully reconciled (either automatically by the system, or by a user) when communications are reestablished and network returns to normal operation [16].

It is necessary to realize that network partitions are not due exclusively to communications failures or site crashes. Networks can be interrupted for tactical reasons (as when a warship decretes radio silence to avoid enemy detection of radio waves) or simply for economical reasons (a corporation batches messages to be transmited over different periods of time to attain lower communications costs).

The goal of this thesis is to analyze and evaluate some of the proposed methods for dealing with the network partitioning problem and to give some useful ideas towards the solution of this problem, especially when availability is a prime consideration in the design of the system.

Chapter 2 presents some basic concepts that will be useful for a better understanding of the following discussion and also presents some problems and issues that will arise when the network partitions. Chapter 3 presents a

survey of methods proposed to deal with network partitions, placing special emphasis on two of them that allow non-stop operation of partitions. In chapter 4 we present an alternate approach to continuous operation of partitions based on precedence graphs. We also present the algorithm required to detect conflicts and reconcile the database at network reconnection time.

II. THE NETWORK PARTITION PROBLEM

A. INTRODUCTION

The best way to describe the problem presented by network partitions is by giving an example. Suppose we have a network composed of three nodes A,B and C, and nodes A and C have a copy of data-object X, which may contain for example a record of the savings account of certain person in a Bank. Suppose that the communications are interrupted in such a way that site A can communicate with site B but none of them can communicate with site C and thus dividing the network in two partitions P1 (formed by nodes A and B) and P2 (formed by node C). In this case both partitions P1 and P2 have access to data-object X, but if we allow both partitions to independently update data-object X, they may perform inconsistent updates to it. This will happen due to the imposibility of sending update messages through the interrupted communications line.

Now, putting ourselves in the worst case, assume the savings account of the person mentioned above has 10,000 dollars and the person is not very honest. When he knows about the partition he goes to node A and retrieves all the money in his savings account, and immediately goes to node C and does the same operation.

Thus he will have 20,000 dollars in his hands and the bank will have in each partition data-object X with the same value of 0 dollars in the savings account record. When communication is reestablished between the three nodes and reconciliation is done, we will have a negative savings account record for data-object X and the problem of having to recover that money.

B. BASIC DEFINITIONS AND CONCEPTS

In this section we will review the basic concepts that will be needed in the rest of the thesis. A more complete discussion of these concepts can be found in [7], [10], [11].

A distributed database system is a collection of named data-objects. Each object has a name and m values associated with it, where m =< n and n is the number of sites in the system. The sites are interconnected by a network and each site runs two software modules: a transaction manager (TM) which supervises the execution of transactions; and a data manager (DM), which processes read and write operations on the data stored at the site.

A logical database is a set of logical data-objects. A copy of a logical data-object stored at a site is called a physical data object. Logical data-objects will be denoted by uppercase letters i.e. X, and physical data-objects will

be denoted by lowercase letters i.e. x,...xn. The set of all physical data-objects stored at a site is called the database of that site.

Operations on data are grouped into transactions. A transaction is a program that accesses the database by issuing read and write operations on logical data-objects. In the read case its TM selects one copy of the data-object and issues a read operation to the DM that manages that object. In the write case the TM issues a write operation for every physical copy of the logical data-object. Transactions are the units of consistency and recovery. They can be viewed as larger atomic actions on the system state which transform it from one consistent state to a new consistent state. Transactions preserve database consistency because if some atomic action of a transaction (i.e. a Read) fails then the entire transaction is undone returning the database to a consistent state.

A transaction is made atomic by use of a commit protocol. A commit is an unconditional guarantee to execute the transaction to completion, even in the event of failures. An abort is an unconditional guarantee to back out the transaction. The problem of guaranteeing transaction atomicity in a distributed system is that of insuring that all the sites either unanimously abort or unanimously commit. After the commit the new value is made available to all other transactions.

Concurrency control is the activity of coordinating transactions that access a database concurrently. The goal is to prevent concurrent transactions from interfering with each other, so that every transaction sees a consistent database state. Inconsistencies may arise because transactions, which are the user's atomic operations have a coarser granularity than actions on objects which are the atomic operations directly supported by the underlying system. If several transactions execute concurrently, their actions get interleaved in an arbitrary way, allowing data inconsistencies to arise. Concurrency control mechanisms typically use locks to regulate access to shared resources. The lock is a serialization mechanism which insures that only one transaction at a time is using a specific object. The lock notifies other transactions that the object is currently being used and protects the requestor from other transactions trying to modify the object.

A formal definition of database consistency is based on the notion of a serializable schedule. A schedule is any sequence of actions performed by a set of transactions on database objects. A schedule is serializable if it is equivalent to a serial schedule, that is, to a schedule in which transactions execute serially, one after the other with no concurrency.

A schedule is consistent if and only if it is serializable. Generally serializability is obtained by requiring

that each transaction in the schedule be two-phase and well formed. A transaction is two-phase if it never locks any data after releasing some lock. It is well formed if it always locks in exclusive mode any data that it writes and locks in shared mode any data that it reads. In order to facilitate easy recovery it is required that all the locks be released at the end of the transaction.

A log (sometimes called audit trail or journal) is a history of all the actions of transactions on recoverable objects. Each action which modifies a recoverable object writes a log record giving the old and new values of the updated object. Read operations need generate no log records, but update operations must record enough information in the log so that given the record at a later time the operation can be completely undone or redone. These records will be aggregated by transaction and collected in a common system log. The log is desirable because we want to be able to commit or undo updates in a per-transaction basis without affecting other transactions.

C. PROBLEMS AND ISSUES

In this section we present some problems and issues that should be considered when dealing with the problem of network partitioning.

1. Alternatives for System Operation Under Network Partitions

When a network partition occurs we have three basic alternatives:

(1) Halt all transaction processing in the partitions until the network is completely reconnected again.

(2) Allow one partition to process transactions that update data-objects while the rest may accept read-only transactions.[1]

(3) Allow all partitions to continue operating "in parallel" during partition and reconcile the databases at partition merge.

We could consider two more alternatives. First, to delay all transactions during the partition, and second, to execute all transactions and then roll-back the entire data-base reexecuting again all transactions after partition ends. These alternatives are not considered because we would be better off if we simply use alternative (1). Clearly alternative (1) is not reasonable since we have as one of the advantages of a distributed system its increased availability. Halting transaction processing in all partitions will be contrary to the idea of having replicated data to make data accessible after failures.

---

[1]The user should receive a warning which alerts him of the possibility that the values may be out of date.

Alternative (2) seems more practical and is in fact usually taken as a reasonable compromise. Most of the methods proposed to deal with network partitions allow only one group of sites to process transactions [1],[15],[9] . Allowing only one partition in operation facilitates recovery after the partition since to reconcile the databases it is only required that sites in the non-active partitions perform all the updates they missed. The only problem in this approach is to guarantee that at most one group of sites processes transactions. In chapter 3 we review some of the methods proposed in order to achieve this objective. However, these approaches may be unacceptable to those sites that must remain non-active during partition when availability is highly desired.

The third alternative, to allow all partitions to process transactions, should be the goal of a distributed system where availability is one of the primary concerns. However, these are some serious problems in allowing "parallel" operation of partitions. As each partition processes different transactions and stores different values into the databases, the values of the data-objects stored of sites in different partitions will diverge and database reconciliation is required when the network is reconnected.

In order to make the databases consistent after partition we can use two strategies. The first strategy is to undo transactions that made conflicting updates to data

objects. For example, assuming two two partitions, at partition merge transactions in different partitions that updated physical copies of the same data-object are detected and some of them are undone. The value of the data-objects in the partition where the transactions were undone is made equal to the value updated by transactions that were left. An important consideration is that each transaction that read the values updated by a transaction that was undone, should be also undone. This requires a detailed log and the necessary overhead to detect conflicting transactions. Also the users that executed transactions during partition will not know if the values produced by their transactions are valid or not until partition is corrected.

The second way of achieving mutual consistency after partition is to use semantic knowledge in order to "integrate" the values of diverging data-objects [19]. This is a very difficult problem and has been discussed in detail by Faissol[2] in [8]. For example, an object r in an airline reservation system indicates the number of available seats in a flight. If after the partition values of object r are v1 and v2, then the correct value of r is given by v1 + v2 minus the value of r before the partition. Note that if the reconciled value is negative then reservations will have to be cancelled with the consequent discomfort of some

---

[2]By the use of partitionable integrity assertions. This is discussed in chapter 3.

affected customers. Obviously, special measures should be taken in partitioned mode operation to avoid these problems.

As we can see we have to pay a high price in order to assure increased availibility of data during network partition. However, there are some circumstances that lessen the overhead which would be incurred in detecting and solving conflicts otherwise. For example, it has been pointed out in [5,6] that in a large class of applications most transactions require little or no synchronization at all because they will never interfere with each other.

2. Correct Operation Under Network Partitions

In order to provide correct operation of a distributed database under a network partition there are three aspects that should be observed:

(1) Preservation of mutual consistency.

(2) Compliance with integrity constraints.

(3) Control of external actions.

Within each partition mutual consistency between copies of data-objects at different sites is preserved using concurrency control methods in the same way as they would be in a connected network. Therefore, each copy of the database in different partitions is internally consistent. However, since there is no communication between partitions the transactions in each partition will run without coordination between them and we may end up with different values

of the modified objects. That is, if the same logical[3] data-object is modified by transactions in different partitions, then we will have a globally inconsistent state. Note that even if the value of the same data-object in two partitions is equal we cannot assume that the correct value is the value stored at both partitions. For example if our bank account balance is 5000 dollars in each partition and it is debited equally with 2000 dollars we will find at partition merge that both balances are 3000 dollars. However, this is not the correct value since if both transactions would have been executed with a connected network the final value of the account balance would be 1000 dollars.

Assuming we do not know anything about the semantics[4] of updates applied to data-objects we can solve the inconsistency that arose in the example above, at partition merge by first, detecting the conflicting transactions in both partitions and second, reconciling the two copies of the data-object. Reconciliation will require that one of the transactions be backed out, then forward the update of the remaining transaction to the other partition and finally to execute the backed out transaction in both partitions. Figure 2.1 shows the process.

---

[3]See definition in section B.

[4]As is the case on the method we develop in chapter 4.

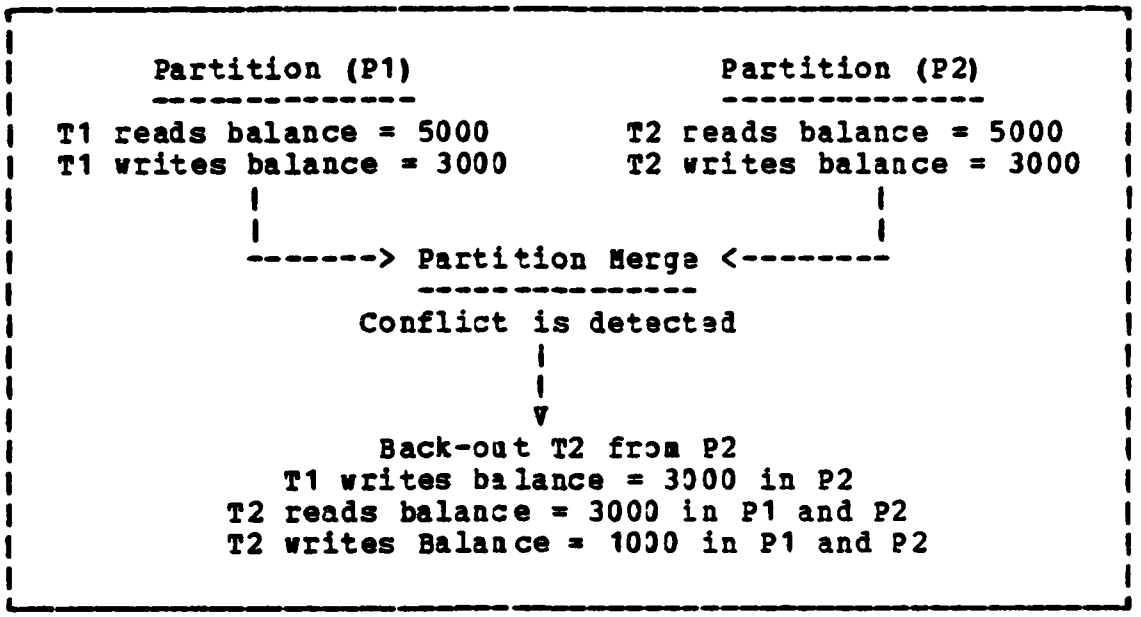


Figure 2.1 Restoring Mutual Consistency

Of course, it is not always this simple, and we will have to make several considerations to restore mutual consistency in more complicated cases. However, the main idea behind this example is that it is possible to restore mutual consistency in a database that has been idependently modified in different partitions and so mutual consistency can be preserved.

Integrity constraints had been classified in [2] as operational constraints and semantic constraints. Operational constraints are those related to the preservation of database integrity against inconsistencies that arise from the concurrent execution of several transactions.

As we have seen the concurrency control mechanism will assure that operational constraints are not violated. Semantic constraints are those related to the preservation of the database integrity against inconsistencies that arise from violations of what data is supposed to mean. For example, in a record for a course containing fields: Exam%, Homework % , Labs % indicating the percentage of the grade devoted to each of them, we would expect that the sum of the values of the fields is 100.

Unless we use semantic knowledge to implement an approach to continuous operation of partitions as in [8], the requirements for compliance with operational and semantic constraints are the same in each partition as the ones in the completely connected network.

In addition to the database contents, external actions may have been performed in response to a transaction and some of these cannot be reversed. For example dispensing cash to a customer is in theory an irrecoverable external action. Under a network partition the problem of allowing external actions becomes more complex because of the independant execution of transactions in different partitions. External actions must be restricted when operating in partitioned mode unless we can reverse the external action by some kind of compensation. For example a message send to a terminal must be followed by a validation note. If the validation note is not received then the user will

know that the message received may not be valid and decide what action to take. If we cannot give a compensation for an external action then it should not be allowed. For example we should not allow cash dispensing since we cannot compensate for it. However, one of the partitions may be allowed to execute external actions provided that this action is not repeated in other partitions at partition merge.

III. PREVIOUS WORK ON PARTITIONING

A. INTRODUCTION

In this chapter we will review previous methods proposed to deal with network partitions. As we saw in chapter 2 the most used alternative was to allow only one partition to update the database. All other partitions were to stop the updating activity on their databases in order to facilitate the database reconciliation at partition merge.

Section B presents a very brief review of some of these methods. We do not spend too much time analyzing them because availability is significantly restricted and we are more interested in continuous operation of the different partitions under partitioning. Also none of these approaches openly states how conflicting versions of data-objects are detected or what is to be done with them upon partition merge.

We are specially interested in high availability of data, so the methods presented on section C which allow non-stop operation of partitions will be presented in more detail.They are two recently proposed methods, the first one uses the version vector mechanism in order to detect file conflicts. This approach is more suitable for an operating system environment. The second approach is based on semantic

knowledge about operations on the data stored in the distributed database.

B. APPROACHES INVOLVING ONE OPERATING PARTITION

1. Voting

There are some proposed Voting based systems in the literature [15], [9]. There are two ways to implement these Systems. The first one is more suitable for fully replicated databases. Here each site is assigned a weight (a number of votes). When a partition occurs the sites in the partition with a majority of votes are allowed to process the transactions (i.e. update data-objects). Sites in other partitions go down or are allowed to process read-only transactions. The advantage of this approach is that if a user has access to a site which is up he has access to the entire database, however users that have access only to down sites are restricted to read the data only.

The second implementation [9] is more general in the sense that it does not require a fully replicated database. The users desiring to modify an object must lock it by obtaining a majority in a vote. That is, updates are only allowed if a majority of sites vote to allow the update. Since there can be at most one partition containing a majority of sites, any object will be updated in at most one partition. However, it may happen that there will be no

partition which contains a majority of sites so in this case updates are not allowed in any partition.

Consistency is easy to preserve since at partition merge minority sites would receive the missed updates and apply them to their copies in time stamp order. This is a clear example where mutual consistency is guaranteed at expense of availability. A disadvantage of voting in general is that it may be unacceptable for the minority sites to be prevented from operating during a partition.

2. Tokens

In this approach it is assumed that each data-object has a token which can be passed from copy to copy. Only sites in the partition containing the token are permitted to modify the object. In other words if the token for every data-object accessed by a transaction resides at some site in a given partition then the transaction may be executed in that group, so using tokens might be less restrictive than using voting.

This approach seems to be best suited for a file system, where transactions access a single data-object. The problem of having transactions accessing more than one data object is that there may be transactions that cannot be executed at any site since the necessary tokens are in different partitions. Also a disadvantage is that tokens can be lost (i.e. in a hard crash), and the problem of

recreating tokens is nontrivial. Furthermore there is a danger of making a resource unavailable if when the partition occurs the token was in a very rarely used part of the network.

3. Primary Sites

This method was originally proposed in [1]. In this approach each data-object has a primary site which is a single site that is to be appointedresponsible for an object's activities. Transactions are executed in a partition if it contains the primary sites of all the objects in the read and write sets of the transaction.

This approach may provide better availability (similar to the token approach) than the voting scheme, but also suffers from the some problems with respect to sites in other partitions, that is, it may be unacceptable for them to operate without updates.

Note that the idea of primary sites and tokens is the same, but in this case the "token" cannot move around and thus cannot be lost. However, the token approach offers more flexibility because the "primary site" may vary dynamically as required. Also a disadvantage of the primary sites approach is that upon partitioning if a primary site was involved in a site crash and a backup site is elected as the new primary site then consistency problems can arise since the information stored in the original primary site is not available to the backup site (i.e recent updates).

4. Reliable Networks

This approach was adopted in SDD-1 [5]. In this system all possible transactions are divided into classes with variable synchronization levels. The classification is made a priori and requires the knowledge of the allowed operations and their semantics. Conflicts of transactions of the some class are avoided by a technique called pipelining based on the assumption that in most applications the operations in a database are known a priori and that most of them do not conflict. What pipelining does is to allow only one transaction of each class to execute at a time in a global time stamp order.

Communications in the SDD-1 are based on the use of a "reliable network" [12], which guarantees that messages are going to be delivered eventually, even when a partition occurs. Messages are saved in "spoolers" to be transmitted following a break in communications. In the case of a partitioned network, nonconflicting classes can clearly operate, but the solution for conflicts within classes clearly can't be implemented due to the lack of communication among partitions, which prevents the exchange of messages necessary to pipeline the transactions. Thus no guarantee of post-partition consistency exists because nothing is done to prevent conflicts between transactions when the partitions merge.

C. APPROACHES INVOLVING MULTIPLE OPERATING PARTITIONS

1. Version Vector Mechanism

This approach was first presented in [16] and was used in the design of LOCUS, a local network operating system at UCLA. It was intended for automatic detection of mutual inconsistency between files upon recovery from network failures and specially upon partition merge. However, the results did not generalize to transactions that accessed more than one file.

Parker and Ramos [17] extended the "version vector" mechanism originally used to implement this approach so as to detect inconsistency when more than one file is used by a transaction. Is important to note that this approach is intended primarily for an environment where file updated rates are moderate and conflicts occur only rarely. In this subsection we are going to give a detailed presentation of this approach.

a. Preliminary Definitions

In this subsection we present some definitions which are required in order to understand this approach. An origin point OP(f) of a file f is a global unique identifier[s] which is assigned to f when it is created. Although

---

[s]For example the pair (time of creation,site of creation)

f's name can change the origin point remains as an immutable attribute. Note that two files based on a common one can have the same origin point.

A **name conflict** occurs when two or more files from the various partitions have the same name but different origin point. A **version conflict** occurs when two or more files have the same origin point but different names and/or different contents. A **file conflict** is detected after a partition if either a name or a version conflict is detected. To restore mutual consistency file conflicts must be reconciled so that file names again uniquely identify a file.

A **partition graph** G(f) for a file f is a directed acyclic graph (DAG) which is labelled as follows: The source and sink nodes are labelled with the names of the sites in the network that contain copies of file f. Each node can only be labelled with site names appearing on its ancestors and each name in a node appears on exactly one of its descendants.

A **version vector** for a file f is a sequence of n pairs where n is the number of sites that store f. The i-th pair (Si:Vi) counts the number Vi of updates to f made at site Si. A set of version vectors are **compatible** when one vector is at least as large as any other vector in every site component for which they have entries. A version vector

is an encoding of the partial order[6] describing the set of updates made at various sites. Independant updates leading to incomparable versions in the partial order, have incomparable vectors as result. A set of vectors conflict when they are not compatible.

For example, suppose that file f is stored in sites A and B . Initially the version vector associated with f will be <A:0,B:0> every time f is modified in one site the version vector will change accordingly. If f is modified in B then the new version vector will be <A:0,B:1>. The version vectors <A:0,B:1> and <A:1,B:0> conflict because no vector dominates the other.

An **execution graph** G = G(T1,...,Tn) is a DAG with nodes C0,L1,C1,...Ln,Cn,Ln+1 where Li is the lock and Ci is the commit operation of transaction Ti repectively. C0 initializes all files and Ln+1 reads all files[7] The edges of G are pairs where either x=Li and y=Ci or y reads what x writes.

b. Description of the Approach

In the case of multiple file conflicts, version vectors alone are not sufficient to detect conflicts, thus an additional mechanism is required in order to achieve this

---

[6]A partial order is a binary relation which is symmetric and transitive.

[7]These are dummy transactions used to give symmetry to the graph.

goal. Conceptually, non serializability can be detected by means of a precedence graph. A precedence graph is composed of an execution graph of a schedule of operations and all edges formed by operations with intersecting read and write sets. If a precedence graph is acyclic then the execution graph within it is serializable.

A set of files S is put into conflict if there exists an schedule of transactions T1,..., Tn whose execution graph is not serializable and one or more files in S are also in the readset of any of the transactions of the schedule. If S is put into conflict then the version vector sequences for the sets S1,...Sn will be incompatible. Note that the sets S1,...Sn are the readsets of the schedule of transactions T1,...,Tn.

With these concepts in mind if we want to detect file conflicts for f, we must check all transaction sets of files S containing f for serializability errors. A way to do this is to have a log where all the readsets of the transactions that had been executed are stored. An operation called extent (f) is defined to obtain the set of files that are involved with f by some of the readsets stored in the log. In mathematical notation:

extent (f) = (g / (f, g) is in R +)

where R+ is the transitive closure of the relation R and R = ((f1, f2) / there is an S in the log such that (f1, f2) is a subset of S).

In plain notation extent (f) is the set of files that in one way or another are related to f in the readset of transactions stored in the log. For example if the log contains:

Read set (T1) = [f, f3, f4]

Read set (T2) = [f3, f4, f5]

Read set (T3) = [f1, f2]

Read set (T4) = [f3, f6]

Then extent (f) = [f, f3, f4, f5, f6]. This is because the transitive closure implies that since f3 and f4 were related with f in Readset (T1) then any other file related with f3 and f4 is going to be also related with f and so on.

Two important consequences of the extent definition are that a file is put into conflict if and only if its extent is put into conflict and that extent divides the set of all files into equivalence classes. In the example above note that extent of f5 is [f, f3, f4, f4, f5, f6] and thus extent (f) = extent (f5).

Stored values of the equivalence classes and their version vector is all what is needed in order to detect multiple file conflicts. The stored set of classes is called a log filter. The algorithm presented for multi-file conflict detection is as follows (LF represents the log filter):

(1) LF = null

(2) Repeat steps (3) to (4) each time a transaction commits

(3) If the readset of the transaction is contained in some set S' in the LF then attach the version vector sequence corresponding to the files in the readset of the transaction with null vectors as place holders

(4) If S is not already contained in LF, incorporate S and its corresponding version vector sequence to LF using the fast union-find algorithm

(5) To check if a file is in conflict get extent of the file in the log filter and see if it has incompatible version vector sequences. If it has, then return conflict.

Note that instead of keeping a list of sequences of version vectors for every update mode in the system, log filters are used to reduce the number of sequences of version vectors the system needs to store as log information. That is, in order to detect conflicts it is only needed to store those sequences which are not dominated by any other sequence.

The conflict resolution policy presented by Parker and Ramos is based on the notion of a transaction. Any file update operation must be within the transaction (between the begin and end statements). A get statement is defined which informs the system about which files the user plans to use. This get statement will check if all the

files given to it are consistent. If there exist a file in conflict then the transaction is not executed. Each file specified in the get statement is locally locked for the duration of the transaction. When the transaction ends the system updates the log filter and the updates are commited simultaneously. If a file is found to conflict at this moment then it should be rolled-back. The transaction completes and all its locks are released.

The proposed approach might be seen similar to "optimistic" concurrency control [4], [14] where conflicts are detected during and/or after the transactions execution. It could be used for partition handling for these concurrency control mechanisms as follows. When working in a partitioned mode, the users are notified of file conflicts whenever a transaction is started and the partition is being merged or is already merged. Once a file conflict is detected no updates are performed on that file until it is reconciled.

2. Semantic Knowledge

This approach was developed by Faissol [8] and is by far the most complete presentation of a method to deal with the network partition problem. The approach is based on the use of semantic knowledge about the applications in order to allow updates in independant partitions. Database operations are divided into classes of semantics in order to

reduce the amount of semantic information that must be supplied to the DBMS. Each class shares a common merge algorithm and information gathering routines.

a. Semantics of Operations and Database Reconciliation

Semantic information is supplied to the DBMS in three forms:

(1) The class of semantics of each operation.

(2) A set of integrity assertions for each operation.

(3) The program code for each operation.

With this information, the DBMS will appropriately modify the behavior of the user operations under a network partition in a way that guarantees that reconciliation can be made automatically upon partition merge. In this approach the applications programmer will be in charge of extending the requirements for semantic integrity in a way that allows partitioned mode operation. Semantic integrity is provided by a mix of integrity assertions and strong data types. Integrity assertions are used mainly for those constraints that may vary with the ocurrence of network partitions. Two sets of these assertions are specified one for normal operation and one for partitioned mode. They will be automatically enforced by the DBMS depending on the status of the system. Only when irrecoverable external actions are involved it is necessary to restrict user's actions by having more strict integrity assertions.

Operations are divided into five classes of semantics depending on their properties. The first class of semantics (class A) involve operations defined as replace value and update single objects, that is, those operations that update an object without examining the database and have no associated integrity assertions. An example of these operations are the update of names and addresses. The second class of semantics (class B) involve operations that are compressible, commutative, update single objects and have partitionable integrity assertions. A set of operations on an account are compressible since we can replace several credits by only one credit that is equivalent to the rest. Two operations are commutative if the order in which they are executed can be chaged producing an equivalent schedule. For example credit and debit operations on an account are commutative. An integrity assertion is defined as partitionable if we can derive from it a set of integrity assertions, one for each partition, such that if each assertion is satisfied in its respective partition then the original assertion is satisfied at partition merge. The third class of semantics (class C) involve operations that either are commutative and invertible or are commutative and have partitionable integrity assertions. An operation is invertible is there exists another operation which will restore the database to the initial state, that is, to the value it had before the execution of the first operation.

For example the debit operation can be inverted if it is followed by a credit for the same amount. Note that if operation Oi is invertible by operation Oj and is commutative with all operations in a schedule then it is invertible even if there exist some operations between Oi and Oj in the schedule. The **fourth class** of semantics (class D) involve operations that are invertible. Finally, the **fifth class** of semantics (class E) involve operations that do not contain irrecoverable external actions.

As we have seen these semantic classes go from the most simple operations to the most complex. An important restriction that must be mentioned is that the invertibility property of operations implies that no irrecoverable external action may be allowed. In order to store information necessary to the reconciliation algorithm a history type is defined for each class of semantics. The set of all history objects created in one partition is defined as the partition history. A partition history will in general contain objects of various classes. It is created when the partition occurs and it is deleted when all merges are complete. The first three history types are stored as a set, that is no ordering is defined. The two last history types are stored as sequences. Figure 3.1 summarizes the class of semantic of each operation and the information necessary to store in each partition history.

CLASS	TYPE OF OPERATION	HISTORY REQUIRED
A	replace value updates single objects	name of objects
B	compressible commutative updates single objects partitionable assertions	names and initial values of modified objects
C1	commutative partitionable assertions	operation-ID, and parameters
C2	commutative invertible	operation-ID and parameters
D	invertible	operation-ID and names in R/W set
E	no irrecoverable external actions	operation-ID, parameters, names, values in R/W set

Figure 3.1 Semantic classes and histories

As we mentioned before, operations with class A semantics involve the lowest overhead and are the most restricted. Objects can be reconciled simply be choosing the value in one partition and installing it in all others. Note that only the last modification of each object is required to be stored in the partition history for class A operations. Operations of class B have little overhead also. Reconciliation of objects can be made independently

for each object by a single operation that summarizes all the updates made in the other partition. Since it is not required that they be invertible they may contain irrecoverable external actions. Operations with class C semantics can modify several objects at a time and therefore, reconciliation of the database is made on an operation by operation basis. At partition merge, each operation that ran in one partition is executed in all the others. Operations can be executed in any order since they are commutative. The subclass of semantics C1 allows irrecoverable external actions but subclass C2 is not allowed to execute these kind of actions since they are invertible. If some integrity assertion is violated by an operation of class C2 then some operations are inverted until a consistent database is obtained. Operations with class D semantics are more complex since they must be executed in order in all the other partitions at partition merge. In this case conflicts may arise because of integrity assertions violations or because operations that involve the same data-objects were executed in different orders in each partition. To reconcile the database conflicting operations must be inverted, taking care of inverting also operations that read values produced by inverted operations and then reexecuting these operations in all partitions. Clearly the partition merge algorithm is more complex. Operations with class E semantics include all operations except those with

irrecoverable external actions combined with nonpartitionable integrity assertions. To reconcile the database modified by operations in this class it is necessary to undo these operations by restoring the "before" images of all modified objects taking the same precautions as with operations of class D semantics.

It is important to be aware that there exist some type of objects called by Faissol "critical types" which cannot be handled by this approach to partitioned mode operation. For example a bank stop payment order. Failure to handle these kind of cases automatically does not invalidate the method since they are infrequent enough to be handled by extraordinary means (i.e. by telephone).

b. System Operation

This approach assumes that a concurrency control mechanism exists in each partition to handle concurrent execution of transactions. Also it is assumed that a recovery mechanism removes the effects of a system crash from the database. System and applications software are not directly available to the users of the database system, who interact through a set of pre-defined transactions. System operations are added to those supplied by the application in order to enforce semantic integrity and to allow partitioned mode operation. When the entire network is connected the system is in normal operation and all the copies of the

replicated database are mutually consistent. Every time a transaction is submitted, a STATUS operation checks a PART-FLAG object which is a system defined type and contains information about the state of the network. If the result of the check is "network conected", then the user operation is executed. It is followed by a check operation on each of the integrity assertions. If all assertions are satisfied then the irrecoverable external actions are started (if they exist) and the transaction terminates. If the assertions are not satisfied then the transaction is aborted. Note that in normal operation, the only additional overhead is the status operation because it is always required to maintain semantic integrity. If status returns "partition merge in progress" the DBMS must check if the operation can be executed. This depends on the class of semantics to which the operation belongs. For example, operations with class A semantics have to check if the target object is not locked by the merge algorithm, while operations of class E semantics have to check if their read and write sets do not intersect with the read and write sets of remaining operations in each partition HISTORY, in order to be executed. If status returns "network partitioned" the appropriate information is stored in the partition history for the class of semantics of the operation. If the operation is not within one class of semantics allowed to run in partitioned mode then it is rejected, otherwise the operation is

executed and a check of integrity assertions is performed. When two or more partitions merge, a system process performs database reconciliation using the information stored in the partition history for each class of semantics. A different merge algorithm is invoked for each class of semantics.

IV. ALTERNATE APPROACH TO AUTOMATIC CONFLICT DETECTION AND DATABASE RECONCILIATON

A. INTRODUCTION

In chapter 3 we reviewed previous work done on network partitioning. We were particularly interested in analyzing two recently proposed methods by Parker [17] and Faissol [8] because they allow non-stop operation of each partition and reconcile the conflicts at partition merge time.

Since our objective is to attain high availability of data in the distributed database we will concern ourselves in this chapter with the development of an alternate approach, that will also allow continuous operation of the partitions during network partition.

The approach proposed in this chapter relies on precedence graphs in order to detect conflicts [20] and on serializability as the correctness criteria for database reconciliation.

In order to make basic concepts more understandable the discussion that follows assumes that there are only two partitions and that during partition merge all the operations on the database are suspended. In later sections we relax these constraints and present some extensions to the approach.

B. DESCRIPTION OF THE APPROACH

1. Preliminary Definitions and Assumptions

It is assumed that within one partition there is a mechanism to provide concurrency control and atomic transactions. A number of such mechanisms have been described in the literature [3], [11], [15], [18] . Therefore we assume that the system operates as if only one transaction is executed at a time and that rejected transactions have no effect on the database. It is also assumed that if a system crash occurs in the midle of a transaction, the recovery mechanism will remove its effects from the database.

For the rest of the chapter transactions will have the following structure:

(1) A transaction T wishing only to read a logical data object X, executes a Read-Lock X, which prevents any other transaction from writing a new value of X while T is reading. However, any number of transactions can hold a read -lock on X at the same time.

(2) A transaction wishing to change the value of logical data object X first obtains a write-lock for X and no other transaction can obtain either a read or write-lock on the object.

(3) Messages are sent to all sites holding physical copies of data-object X notifying them to change their copies to reflect the new modification before releasing the write-lock.

(4) The transaction commits and only then all its locks are released (Thus we assume two-phase locking to assure serializable execution).

Definition 4.1: The logical data objects* that a transaction read is its readset. The logical data objects that a transaction writes is called the transaction's writeset. They will be represented as readset(T) and writeset(T) respectively.

Note that in particular we do not assume that the writeset of a transaction is always a subset of the readset. This allows a more realistic model which admit the possibility that a transaction reads a set of objects (the readset) and writes a set of objects (the writeset), with the option that an object X could appear in either one of these sets or both. For example in the transaction:

READ X; READ Y; Z = X * Y; X - Y; write Z; write X

the readset is X, Y and the writeset is X, Z .

Definition 4.2: A precedence graph G(V,E) is a directed graph, where the vertices (V) correspond to the set of transactions T1,...,Tk within a schedule S, and the edges (E) represent precedence relations between the transactions.

---

*See Chapter 2, section B.

Definition 4.3: A schedule S for a set of transactions T1...Tk is serializable if its precedence graph is acyclic.

Proposition 4.1: Two transactions Ti and Tj are commutative if:

1) READSET(Ti) is disjoint with WRITESET(Tj) and

2) WRITESET(Ti) is disjoint with READSET(Tj) and

3) WRITESET(Ti) is disjoint with WRITESET(Tj)

Proof Outline: The only way in which transaction Ti may affect the outcome of Tj is by modifying shared objects in the database and viceversa. Since only the readsets are allowed to intersect and read operations are commutative (the order in which transactions read a shared object is unimportant) there is no real interaction between the transactions. Therefore changing the order of execution produces an equivalent schedule, which implies commutativity.

Definition 4.4: Within one partition schedule we define a transaction Ti to be a descendant of transaction Tj if READSET (Ti) intersects WRITESET(Tj).

Definition 4.5: The relatives of a transaction T is the set of all transactions that functionally depend on T (i.e. the set of all descendants).

In order to verify the correctness of the approach given in the next section we need a formal definition of correct partitioned mode operation. We will adopt the definition given by Faissol [8].

Definition 4.6: Let S0 be a schedule composed by transactions Ti...Tk, such that some transactions in S0 were succesfully applied in partition 1 and the rest in partition 2, resulting respectively in the schedules S1 and S2, then correct partitioned mode operation is attained if the following conditions are satisfied:

(1) With the information stored in each partition it is possible to construct schedules S3 and S4 such that schedules S5 and S6 are both equivalent to the same serial execution of S0, where: S5 = (S1, S3) and S6 = (S2, S4).

(2) No transactions containing irrecoverable external actions are reversed by the partition merge algorithm.

(3) All integrity assertions are satisfied after the partition merge algorithm is executed.

Note in particular that only a schedule equivalent to some serial execution of S0 is required and not a schedule equivalent to S0. This may cause different results than would have occured if the network was connected, but this is usually accepted if serializability is the correctness criteria.

Also it is important to note that since some transactions that would have been executed with the network connected must be rejected in partitioned operation, S0 was defined as the schedule of transactions successfully

executed in partitioned mode (this does not mean that all transactions are commited since some of them may be aborted after execution because of violation of some integrity assertion).

2. Conflict Detection and DataBase Reconciliation

Our approach to continuous operation under a network partition is based on the use of precedence graphs to detect conflicts between partitions at merge time and to help to determine a serializable schedule equivalent to some serial execution of the global schedule SO defined in the last section. When a network partition occurs the DBMS within each partition performs two actions: first, activates a mechanism that aborts transactions trying to execute an irrecoverable external action and second, creates a partition-log which stores information necessary for the reconciliation algorithm. The information contained in the partition-log consists of the transaction-ID, read and write sets of the transaction and the old and new values of the updated objects (those in the write set). The transactions are recorded in the order in which they commit* within the partition, that is, as a sequence (a total order). When communication between partitions is reestablished no more

---

*Note that Tn can execute in a partition only if there exist copies within the partition for every data-object in its read and write sets.

transactions are allowed to be processed until partition merge is completed (this restriction is relaxed in section C). The partition merge algorithm is then started to reconcile the databases.

Initially the partition reconciliation algorithm will construct for each partition a precedence graph from the information contained in the respective partition-log. The precedence graph is constructed as follows:

(1) If transaction Ti reads data-object X, and Tj is the next transaction (if it exists) to write in X then construct an edge from Ti to Tj.

(2) If transaction Ti writes data-object X and Tj is the next transaction to write X then construct an edge from Ti to Tj.

(3) If transaction Ti writes data-object X and Tj reads X before any other transaction writes X then construct an edge from Ti to Tj. Mark this edge as a descendant edge.

It is important to note that the partition precedence graph does not have to be constructed at partition merge time, but can be constructed gradually as new entries are added to the partition-log. In fact, it is better to do it this way since at network reconnection the precedence graph will be almost complete and partition merge time is reduced. Also note that each partition precedence graph is going to be acyclic since the schedules stored in the

partition-logs are serializable(they had already been executed).

The next step is to construct a global precedence graph which is going to consist of each partition's precedence graph plus conflict edges between partitions. Since transactions were allowed to run "in parallel" in their respective partition without coordination between them, the conflict edges represent the interaction among transactions from different partitions. Therefore, a transaction that reads a data-object in one partition must precede any transaction that writes that data-object in the other partition to mantain consistency. So a conflict edge from Ti to Tj is constructed if transaction Ti in one partition reads or writes data-object X and transaction Tj in the other partition writes X.

Once the global precedence graph is constructed a topological sort is executed on the graph and if a cycle is found, one of the transactions involved in the cycle (the one with less descendants) and all of its descendants are rolled-back in the partition where they were executed. The entry in the partition-log corresponding to each rolled-back transaction is send to a re-execution list. If a node can be extracted by the topological-sort then the values of the objects updated by the transaction represented by the node

are forwarded[10] to the other partition and the corresponding entry in the partition-log is deleted. The process is repeated until all transactions in the precedence graph had been forwarded to the other partition or send to the re-execution list. That is we have no more entries in the partition-logs.

The transactions in the re-execution list are then executed in both partitions and if any violation of integrity assertions occurs the transaction is rolled-back and its entry in the re-execution list is deleted. This can happen because we have altered the order in which non-commutative transactions were executed. When the algorithm terminates we are going to have a consistent database throughout the network.

After the brief discussion of the approach taken we are now ready to present the merge algorithm.

Algorithm MERGE

(1) Send message "partition merge in progress" to each partition.

(2) Construct the precedence graph for each partition extracting information from their respective partition-log.

(3) Repeat steps (4) to (5) for each partition.

---

[10]Actually, only the updated values of copies of data-objects that also exist in the other partition are forwarded. We will refer to this every time the word forwarded is used.

(4) Repeat step (5) for each entry in the partition-log starting at the first one.

(5) Compare the readset of this entry with the read and write sets of every entry in the partition-log of the other partition. Any time a match is found add a directed edge (from the transaction in this entry to the transaction in the other partition's entry) to the global precedence graph. Mark this edge as a conflict edge.

(6) Run the TOPOLOGICAL-EXEC algorithm on the global precedence graph until all nodes on the graph had been deleted.

(7) Execute algorithm RE.

(8) Send message "merge completed" to each partition.

(9) Terminate.

At end of the merge algorithm the global precedence graph and all entries in both partition-logs will have been deleted. We now present the supporting algorithms topological-exec and RE. What the TOPOLOGICAL-EXEC algorithm basically does is a topological sort on the global precedence graph to obtain a serial schedule for the transactions in both partitions. A topological sort generates a linear ordering with the property that if Ti is a predecessor of Tj in the graph then Ti precedes Tj in the linear order. A linear order with this property is called a topological order [13].

Since a linear order is serial by nature a topological order gives a serial order which satisfies the precedence relations between transactions.

It is important to note, however, that a topological order can be obtained only if the global precedence graph is acyclic and thus if there is a cycle it must be removed from the graph before the topological sort can continue. Algorithm TOPOLOGICAL-EXEC uses algorithm REMOVE-CYCLE in order to remove one of the edges that form the cycle to obtain an acyclic graph. Every time the TOPOLOGICAL-EXEC algorithm is able to extract a node from the graph it forwards the updates made by the transaction (contained in the partition-log entry) that corresponds to the graph node to the other partition. We now present the algorithm.

Algorithm TOPOLOGICAL-EXEC

(1) Repeat steps (2) to (6) for each node in the global precedence graph.

(2) If every node has a predecessor then execute algorithm Remove-Cycle and go to (1).

(3) Pick a node which has no predecessors.

(4) Forward the updated values of the data-objects modified by the transaction (contained in the partition-log entry) that corresponds to the selected node to the other partition.

(5) Delete the entry from the respective partition-log

(6) Delete the node and all edges leading out of the node from the global precedence graph.

(7) Terminate.

As we indicated before, any time a cycle is found the algorithm Remove-Cycle is invoked to remove one of the nodes involved in the cycle. This means that the effects of the transaction contained in the entry that corresponds to the node and the effects of all the relatives of the transaction must be removed from the database. In order to avoid extensive roll-back of transactions as much as possible the transaction chosen to be removed will be the one with less relatives. The transactions will be rolled-back in inverse order of execution and their entries in the partition-log will be moved to a re-execution list to be executed again later. We now present the algorithm.

Algorithm REMOVE-CYCLE

(1) Repeat step (2) for each node related to another by a conflict edge in the precedence graph.

(2) Compute the number of relatives of the node by counting the descendant edges that go out either of the node or its descendants.

(3) Choose the node with less number of relatives and create a relative set containing all the relatives of the node. If there is more than one node with the same number then choose the one involved with more conflict edges.

(4) Move the partition-log entry corresponding to the selected node to a roll-back list and repeat step (5) for each following entry until the relative set is empty.

(5) If the entry corresponds to a node in the relative set then move the entry to the roll-back list and delete the node from the set.

(6) Repeat steps (7) to (9) for each entry in the roll-back list starting with the last entry and going backwards until the list is empty.

(7) Use the system supplied UNDO operation to remove the effects of the transaction, corresponding to the entry by placing the "before" values of the updated objects in their correspondent partitions.

(8) Move the entry in the roll-back list to the re-execution list.

(9) Delete the node that corresponds to the entry and all edges to or from the node in the global graph.

(10) Terminate.

At the end of algorithm TOPOLOGICAL-EXEC there is a re-execution list which contains all the transactions from both partitions that were rolled-back in order to mantain a global consistent database state. These transactions are to be rerun in both partitions by the algorithm RE. In this case integrity violations can occur since we have changed the execution order of transactions that are noncommutative.

Note that when algorithm TOPOLOGICAL=EXEC terminates the databases of each partition are in a consistent state, that is, they are maintaining mutual consistency since the same transactions had been executed in both partitions. We now present algorithm RE.

Algorithm RE

(1) Repeat steps (2) to (5) for each entry in the re-execution list.

(2) Run the specified transaction in both partitions.

(3) If any integrity assertion is violated reject the transaction.

(4) Delete the current re-execution list entry.

(5) Terminate.

We proceed now to show the correctness of the approach.

Proposition 4.2: Algorithm MERGE correctly reconciles a database that has been independantly modified by transactions in different partitions.

Proof: Let S0 be the schedule in the whole system with S1 and S2 executed in partition 1 (PR1) and partition 2 (PR2) respectively. We must prove that each of the requirements for correctness defined in section B, subsection 1 are satisfied when algorithm MERGE is executed. In order to make the proof more understandable we consider three cases according to the initial configuration of the global precedence graph constructed by steps (1) to (5) from algorithm MERGE.

Case 1 : global precedence graph with no conflict edges. This means that all transactions in one partition are commutative with all transactions in the other partition. Step (6) executes algorithm TOPOLOGICAL-EXEC which will obtains a topological order of transactions from both partitions and will forward the values of objects updated by transactions in one partition to the other. In this case the resulting schedules of transactions executed in PR1 and PR2 will be equivalent to the global schedule S0 and not only to a serial execution of it. This is due to the fact that transactions in PR1 are commutative[11] with transactions in PR2 and viceversa and they can be executed in any order[12] without affecting their results.

Case 2 : Global precedence graph with conflict edges but without cycles. In this case the graph is also serializable. A conflict edge represents the fact that for the same logical data-object with physical copies in different partitions, a transaction in one partition read the value of a copy of this data-object while in other partition a transaction updated the value of the copies of the data-object.

---

[11]See definition of commutativity in section B.

[12]The order in which they were executed in their own partition must be preserved.

Algorithm TOPOLOGICAL-EXEC by step (3) will make sure that the transactions which read an object forward their updated objects before transactions that write the object in the other partition. The rest of transactions are commutative with the transactions in the other partition so they are no problem.[13] However.the resulting schedule executed PR1 and PR2 may not be equivalent to S0 but only to a serial execution of it, namely, the one produced by algorithm TOPOLOGICAL-EXEC.

Case 3 : Global precedence graph with cycles. In this case the graph is not serializable. Cycles must be removed to obtain a serializable schedule. Step (2) of algorithm TOPOLOGICAL-EXEC will detect cycles and remove all offending transactions and its relatives using algorithm REMOVE-CYCLE. Removed transactions are sent to the re-execution list. Once the graph has no cycles we are again in case 2. Values of objects updated by transactions are forwarded to the correspondent partition in topological order by step (4) of algorithm TOPOLOGICAL-EXEC. Immediately before step (7) of algorithm MERGE, the schedules in both partitions are equivalent with each transaction not removed from the graph executed in both sides and transactions removed from it in the re-execution list. Step (8) will

---

[13]Same as in case 1.

then execute algorithm RE, which reruns the transactions that were removed in both partitions in the same order, checking integrity assertions. If some integrity violation occurs at this point,the transactions are aborted in both partitions. At this stage every transaction in the global schedule S0 except those with integrity violations had been executed in both partitions. Thus, the resulting schedules are equivalent to some serial execution of S0, namely, the one produced by algorithm MERGE. Note that this schedule will be composed by transactions executed in topological order by algorithm TOPOLOGICAL-EXEC, plus transactions rerun by algorithm RE minus transactions with integrity conflicts. Correctness condition (1) is satisfied because in each of the three cases we are going to have at least a schedule equivalent to some serial execution of S0 in both partitions. Condition (2) is satisfied because no irrecoverable external actions are allowed. Condition (3) is also satisfied because transactions that violate integrity assertions are aborted.

As we can see the merge algorithm is somewhat complex. This is due to our interest in allowing more availability of data and to our concern in trying to avoid as much transaction roll-back as possible.

3. An Example

In order to make clear how the approach works we present in this subsection an example.

Let S1 = T11, T12, T13 be the schedule of transactions executed in partition 1 (PR 1) where:

Readset(T11) = x,y,z ; Writeset(T11) = x,z

Readset(T12) = u,v,z,p ; Writeset(T12) = v

Readset(T13) = p,q ; Writeset(T13) = p,q

and S2 = T21,T22,T23,T24 the schedule executed in PR2 where:

Readset(T21) = q,u,r ; Writeset(T21) = u,r

Readset(T22) = l,m,n ; Writeset(T22) = l,m

Readset(T23) = u,w ; Writeset(T23) = w

Readset(T24) = w,y,z ; Writeset(T24) = y

When the partitions find out that they can communicate algorithm MERGE is started. Step (2) of this algorithm will construct the precedence graph of each partition with the information stored in their respective partition-log. Steps (3),(4),(5) of the algorithm will construct the global precedence graph by adding the conflict edges to the existing graph. Figure 4.1 shows the global precedence graph constructed.

Once the global precedence graph is constructed step (6) will call algorithm TOPOLOGICAL-EXEC to obtain a topological order of the nodes in the graph. Step (3) of algorithm TOPOLOGICAL-EXEC will select the node corresponding to T22

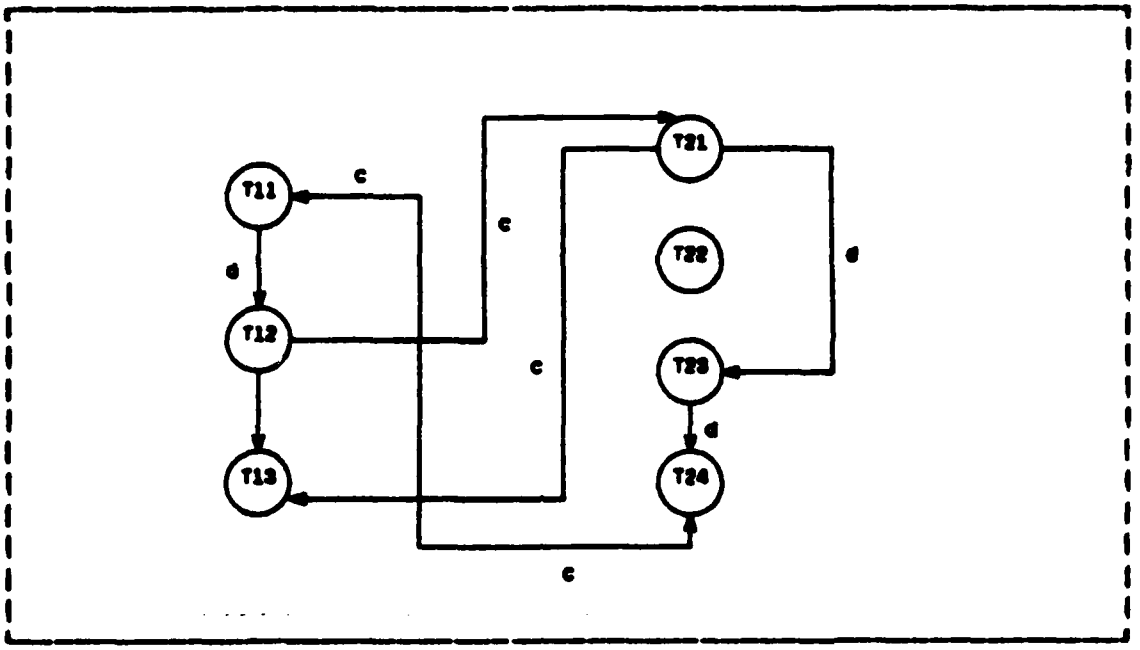


Figure 4.1 Initial precedence graph

since it is the only node without a predecessor. Step (4) of this algorithm will forward the value of the objects updated by T22 (i.e. 1,m) to partition 1 (PR1). Steps (5) and (6) will delete the entry in the partition-log that corresponds to that node and will delete the node from the graph respectively. Figure 4.2 shows the state of the graph after the deletion.

Step (2) of algorithm TOPOLOGICAL-EXEC will determine that all remaining nodes have a predecessor so a cycle exists. Algorithm REMOVE_CYCLE is than invoked and steps (1) and (2) of this algorithm count the descendants of each node

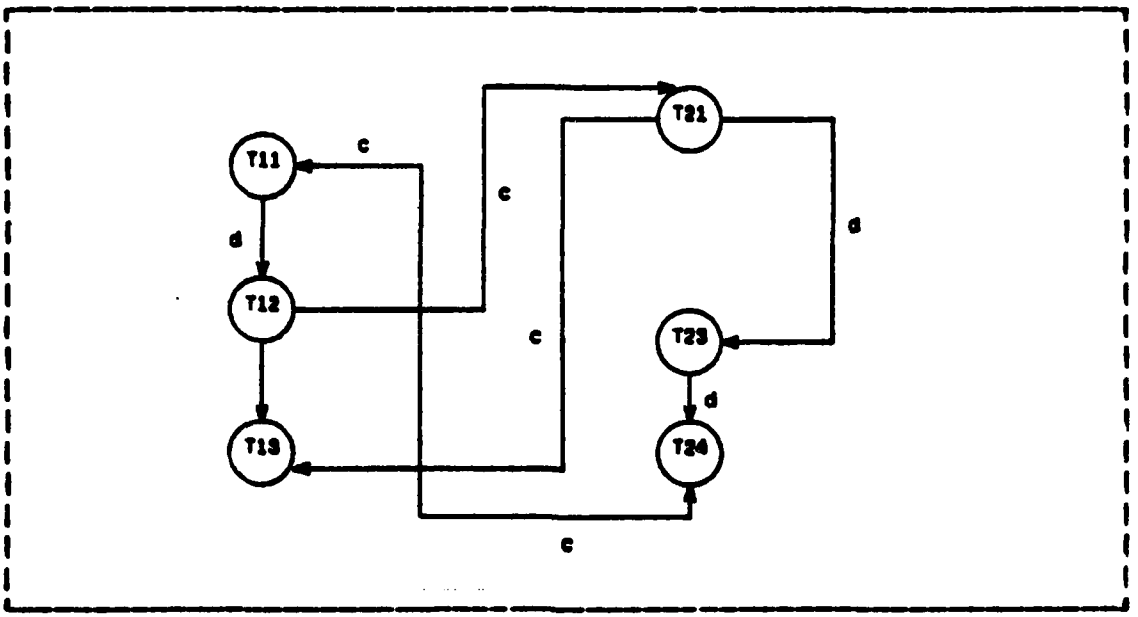


Figure 4.2 First modification to precedence graph

related to another by conflict edges. T12 and T24 have the same number of descendants (0 descendants) but T24 is involved with two conflict edges so step (3) of this algorithm chooses T24 to be rolled-back and creates an empty set of descendants. Step (4) moves the partition-log entry corresponding to the selected node to the roll-back list and step (5) is skipped since the node has no relatives. Steps (7), (8), (9) remove the effects of the transaction from the database by using the UNDO operation, move the entry corresponding to T24 to the re-execution list, and delete the node and edges to or from it from the graph respectively. Figure 4.3 shows the new state of the precedence graph.

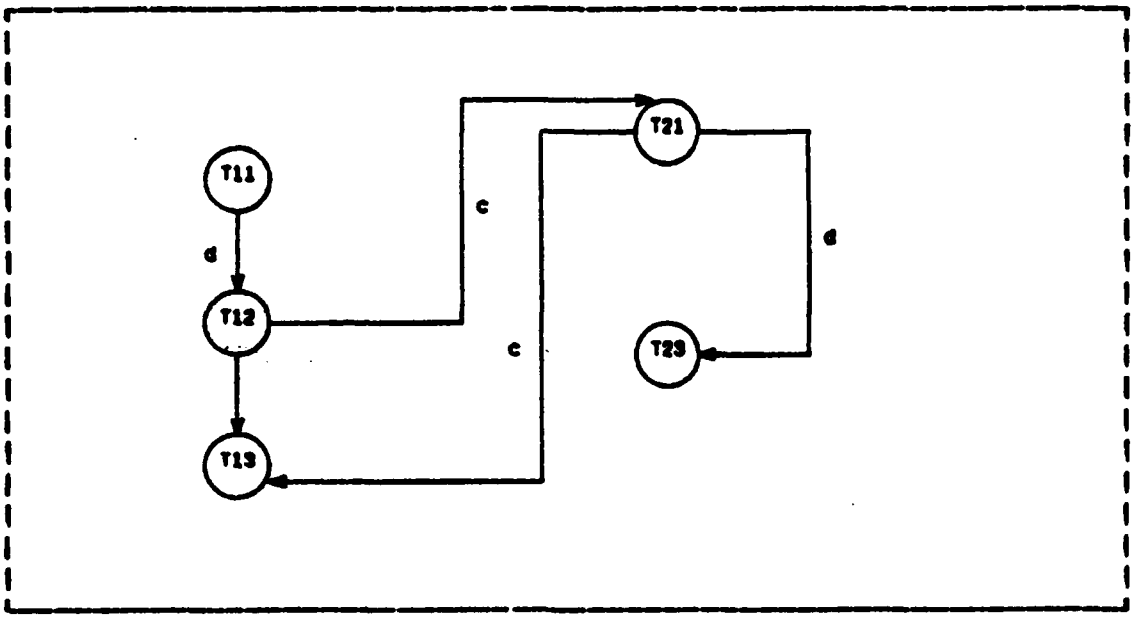


Figure 4.3 Second modification to the precedence graph

Algorithm TOPOLOGICAL-EXEC reassumes execution and by step (3) picks T11 since now it has no predecessors. Step (4) forwards the values updated by T11 to PR2 and steps (5) and (6) delete the entry corresponding to T11 and the node in the graph respectively. Figure 4.4 shows the remaining graph.

Succesive applications of step (3), (4), (5) pick T12, T21, T23, T13 in that order (T23 and T13 could be picked in any order); send the updates of the transactions to the partition where they did not execute and, delete the respective entries from the partition-log and nodes from the graph. Figures 4.5 show the next state of the precedence graph.

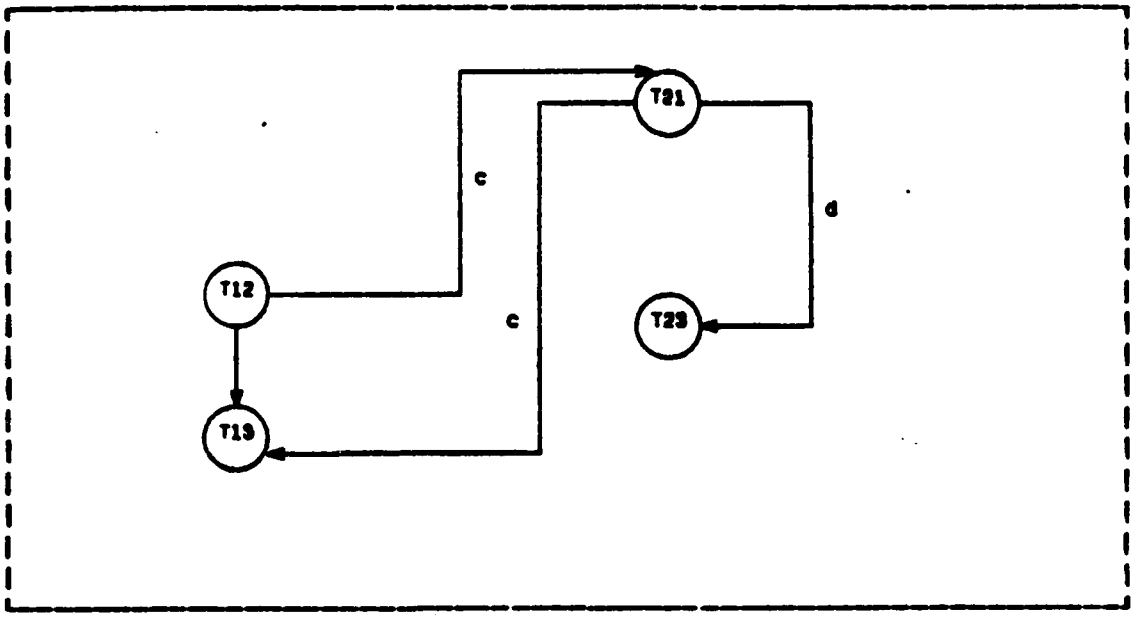


Figure 4.4 Third modification to the precedence graph

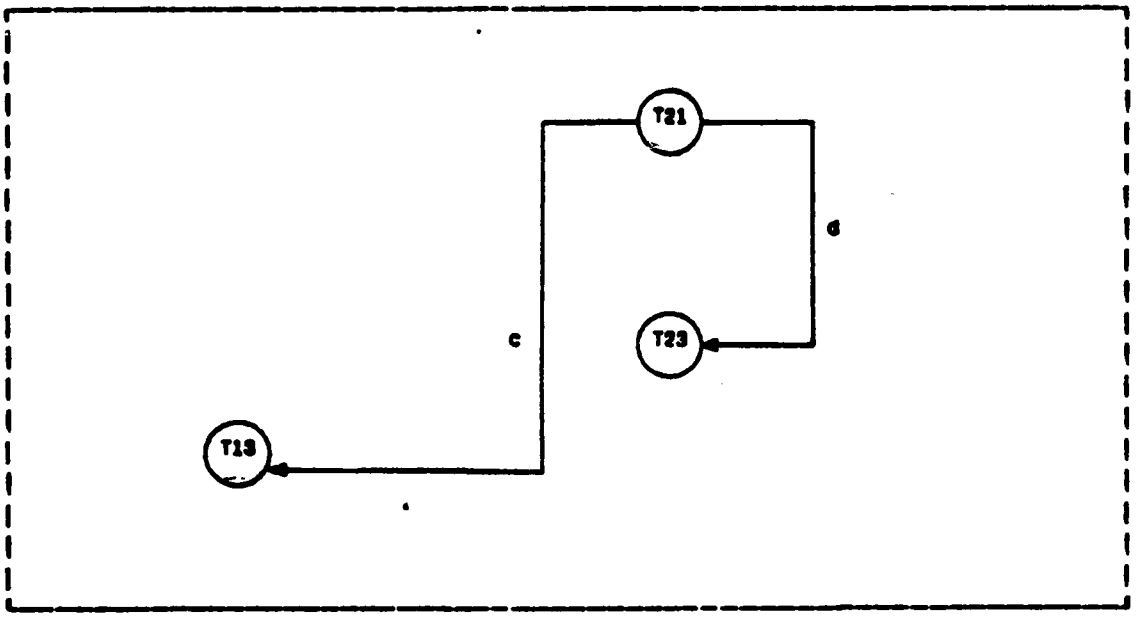


Figure 4.5 Fourth modification to the precedence graph

Now algorithm MERGE reassumes execution and by step (7) it executes algorithm RE. The only entry in the re-execution list was T24 so this transaction will be executed in both partitions.If any integrity assertion is violated the transaction will be aborted by step (4) of this algorithm and by step (5) the entry is deleted from the re-execution list.

Note that the schedule of transactions executed in both partitions is now equivalent to a schedule executed in topological order.This is due to the fact that sending the updates made by any transaction in PR1 to PR2 is equivalent to executing that transaction before any transaction in PR2 that writes in those objects and viceversa. Thus the equivalent topological order of execution in both partitions will be:T22, T11, T12, T21, T23, T13, and T24 if it is not aborted.

C. EXTENSIONS TO THE APPROACH

Section B presented the approach we proposed to allow the operation of distributed database systems under network partitions. In order to simplify the description of the systems operation and of the merge algorithm, we made a number of restrictions and promise to relax them later. This is the purpose of this section.

Subsection 1 presents a discussion of the locking requirements and associate modifications to the merge algorithms that will allow normal operation while a partition merge is in progress. The main objective of this section is to reduce the delay to which incoming transactions would be subjected while the merge algorithm is in progress. This delay can be substantial for partitions of long duration or for database systems with high activity rates.

Subsection 2 presents a discussion of partial partition merges when there are more than two partitions. Sites may become partitioned and then join again in various orders. The easiest solution would be to wait until the network is completely reconnected to perform partition merges. This would not only increase the degree of inconsistency that must be reconciled later but also would increase the overhead and time required for partition merge.

Finally subsection 3 presents a discussion of the situations in which irrecoverable external actions can be allowed when network is partitioned. The main objective of this subsection is to increase user's availability to data in the database, so that a higher number of transactions can be executed during the partition.

1. Normal operation During Partition Merge

The merge algorithm described in section B, subsection 2 assumed that no transaction was allowed until the

algorithm was completed. This assumption relieved us from worrying about interference from other transactions and locking issues in the description of the algorithm.

This subsection presents the locking requirements necessary to allow normal operation while the merge algorithm is in progress. We will see that these requirements are relatively simple, despite the fact that the algorithm is somewhat complex.

Normal operation during partition merge can be allowed if the new transactions do not interfere with transactions being reconciled by the merge algorithm. That is we need the new transactions to be commutative with all the remaining transactions in each partition-log in order to execute them in normal mode. Otherwise new conflicts will arise that could not be resolved.

To assure that the new transaction is commutative we need to compare its read and write set with the read and write sets of all transactions still in the partition-logs. If no match is found then we know it is commutative and can be executed without problem. However if there is a match then the transaction will have objects in its read and write set that are yet to be reconciled and so it must be delayed to avoid new conflicts.

Note that even if the new transaction is commutative there may be a significant delay before it can be executed since we are introducing additional overhead in order to

compare its read and write sets with the ones of the transactions that remain to be reconciled.

However, we can attempt an optimization if we use the idea of a data-object log (DO log) [4] to store information about the status of the object. The information we need to store is just a "mark" that indicates that the object was used by some transaction in partitioned mode. In order to accomplish this we need to establish the policy that while in partition mode operation the first time an object is used by any transaction a Data-object-log is created and a value (i.e. 0) is stored in it. After this every time a transaction uses the object the value in the DO log is incremented (i.e. by 1), this process stops when the partition merge algorithm initiates its execution. A small modification should be made to the MERGE algorithm. Every time an entry is deleted from a partition-log or from the re-execution list, the value stored in the Do log of each object used by the transaction that corresponds to that entry is decremented by 1 in the partition where the transaction executed while in partitioned mode. If when deleting an entry the value stored in the DO log is 0 then the DO log is deleted.

In that way new transactions operating in normal mode that are willing to use an object just have to check in each partition if the object has an associated Do-log and if so then the transaction is delayed until the Do-log of the object is deleted in every partition where it existed.

We can see that the overhead involved is much smaller than the one we need to compare the read and write sets with all of the transactions being reconciled and so the delay should be considerable decreased. Also the overhead imposed by the MERGE algorithm with this addition is not very significant and the additional storage required is rather small.

2. Partial Partition Merges

In the presentation of our approach we assumed that only two partitions existed. However, this may not be the case and although it should be quite infrequent there may be more than two partitions that can join in different orders depending on which communication lines are reestablished first. This section relaxes the assumption that only two partitions exist and discusses how to deal with the problem of partial partition merges.

As we mentioned before a straightforward, but simple minded, solution could be to wait until the network is completely reconnected and then start the partition merge algorithm involving all partitions at the same time. However this solution has serious disadvantages such as having more restricted operation within a partition for more time and having a significantly increased overhead in order to merge all the partitions at the same time.

An alternate solution could be to allow partial partition merges to occur in different orders without any restriction and as soon as two partitions discover that they can communicate between them. This can be done because we have in the partition-log enough information (names and values of objects in read and write sets) to avoid repeated updates to the same object and out of order execution of transactions in different partitions. However this will require an extensive comparison of partition-logs and additional lists to store the entries that were removed in a previous merge in order to see if it is required to remove these entries from a new partition joining the existing partition.

An example can clarify these concepts. Suppose we have the partition graph shown in figure 4.6 . Initially the network partitions forming two groups, the first group contains only N3 and the second group N1 and N2. Each partition is assigned a unique partition-ID which is included in all entries made to their respective partition-logs. Later, another partition occurs resulting in N1 and N2 working separately.

Again a unique partition-ID is assigned to these partitions and every entry to the partition-log from now on will have the new partition-ID. Note that each of the new formed partitions N1 and N2 "inherits" the partition-log entries of the past partitions, mantaining these entries

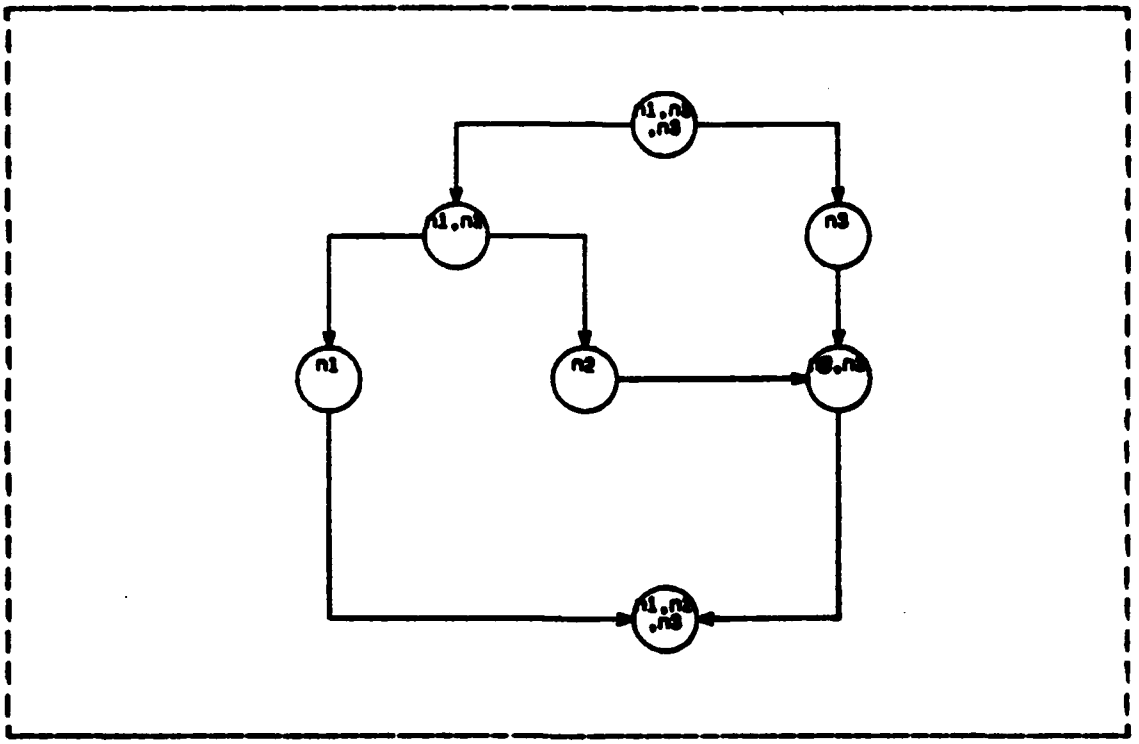


Figure 4.6 Partial merges in a partition graph

their original partition-ID. When N2 discovers that it can communicate with N3 they start the merge algorithm to reconcile their databases. However, no entry of their partition-logs can be deleted since they will be needed to compare new merges if the transactions corresponding to those entries have been executed before i.e. when N1 and N2 were in the same partition. We must also preserve entries in the roll-back lists because if entries corresponding to N1 when it was in the same partition with N2 are rolled-back, then when these two sites are reconnected again we

must roll-back these entries from N2 also. Similar precautions will have to be taken with entries in the re-execution list, where some transactions that were originally executed in (N1, N2) will not be able to be re-executed in (N2, N3) if some object is present only in N1, so those transactions should be delayed in their re-execution until N1 joins (N2, N3). As we can see the required protocols to make possible partial merges in arbitrary orders would be pretty involved and the high overhead would make them impractical. Thus, although it is possible to allow partial merges in different orders we will not pursue this solution because of its complexity.

A far more practical solution could be obtained if we restrict partial merges in such a way as to allow only symmetric merges, that is, if we require that the partition graph be a symmetric direct acyclic graph. Figure 4.7 shows the way in which merges would be executed if we comply with this restriction. As we can see subgrahps are symmetrical, so partitions merge in the same order in which they were partitioned.

Having this merge pattern the only modification we need to introduce in the MERGE algorithm is that we need to retain the entries in the new partition-log. Note that these entries will be stored in topological order, that is, in the order in which the topological-exec algorithm executed the transactions in both partitions. Entries can be deleted when the sink node is reached.

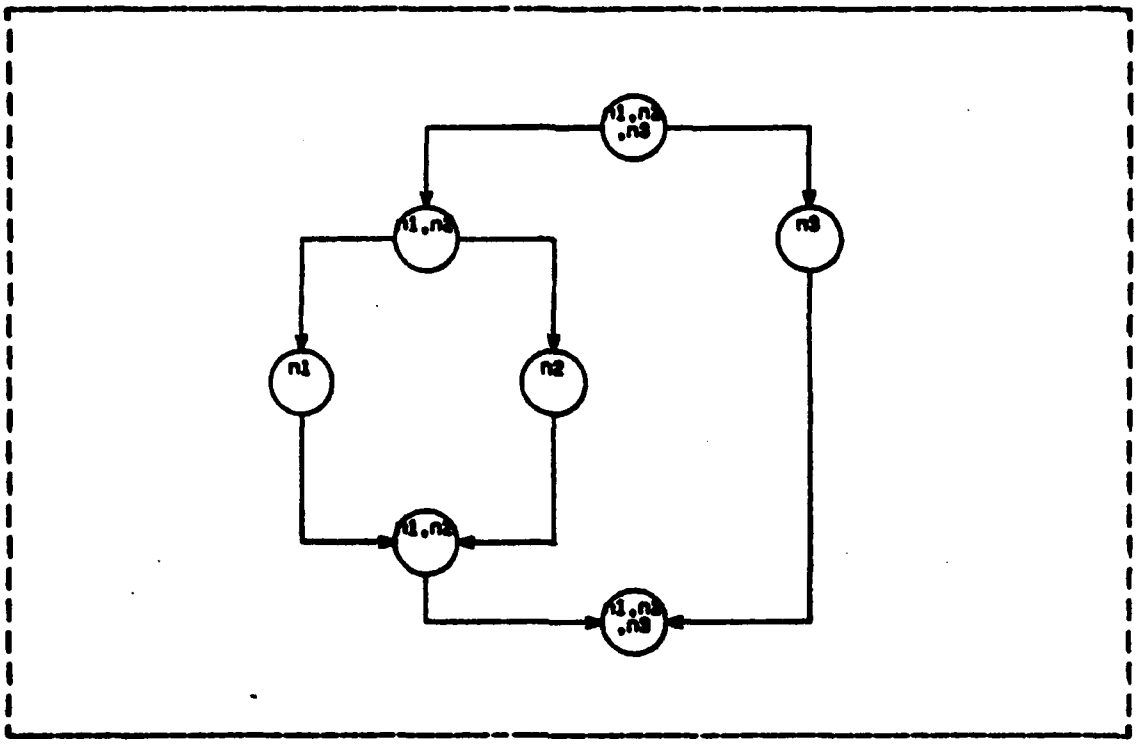


Figure 4.7 Symmetric partial merges in a partition graph

We now present the modifications required in order to allow partial merge in a symmetric directed acyclic graph (DAG). Recall that algorithm MERGE uses algorithm Topological-Exec to delete the entries from the partition-logs so the modification will be in this algorithm. Modification to algorithm TOPOLOGICAL-EXEC.

(5) If the sink node in the partition graph has been reached then delete the entry from the respective partition-log, else store the entry in the new partition-log.

We can now proceed to show the correctness of the modification made to the algorithm.

Proposition 4.4: If the partition graph is a symmetric DAG then partial partition merges are correctly executed by the modified MERGE algorithm.

Proof Outline: Since the partition graph is a symmetric DAG each sub-graph represents a situation which is the same as the one for the original MERGE algorithm. That is, each subgraph will consist of a subsource node and a subsink node that are the same and thus no duplicate updates or out of order execution of transactions may occur. With the modification in step (5) of algorithm topological-exec, partition-log entries are retained after they have been used to reconcile their respective databases. Thus,these entries will be available to be applied in the other subgraphs. There is no problem with transactions that are undone since they will have been executed only in the current subgraph and since no other subgraph was involved they do not need to be undone elsewhere. The same is true for transactions that violate integrity assertions when being re-executed. Thus we have no problem in deleting the corresponding entries from the partition-log. Once the sink node of the entire graph is reached, we have the entire network reconnected and the same situation as in the original MERGE algorithm. When this occurs, the

entries in the partition-logs can be deleted since they will have been applied to all sites.

As we can see this solution requires that partial merges be done in a symmetric way. Obviously this will delay some partial merges since we have to wait until partitions in the same subgraph can communicate between them. As a matter of fact there is going to be more to reconcile afterwards since partitions will continue operating in partitioned mode. However this solution is a compromise between the two solutions that were mentioned first and we think it is a reasonable one.

3. Allowing irrecoverable external actions

One of the assumptions made in section B was that no irrecoverable external actions were allowed and transactions that attempted to execute one of these actions were aborted. In this subsection we anlyze in which circumstances irrecoverable external actions can be allowed.

Faissol [8] proposed a solution to this problem[14] by determining those integrity assertions that could be partitioned in such a way that if they were not violated in any partition then they would not be violated as a whole when the network is completely connected.Thus irrecoverable external actions that involved objects with these type of

---

[14]See Chapter 3,section C.

integrity assertions could be allowed under network partition. This solution can also be implemented in the approach presented in this chapter by giving the DBMS enough semantic information about integrity assertions dealing with external actions.

An alternate solution would be to allow irrecoverable external actions in at most one partition. In fact, there is no reason why we could not do this. In our approach integrity assertions for each partition are the same as the ones when the network is completely connected. The partition to be chosen to allow these kind of actions could be determined by one of the methods proposed in Chapter 3, section B (i.e. voting). Precedence should be given to partitions where it is more likely that external actions will occur. For example if after a partition in a bank system 80% of the automatic teller machines are in one partition, then this partition should be allowed to execute irrecoverable external actions (i.e. cash dipensing). This solution has the advantages that it does not require extra integrity assertions and that users which can access the selected partition will have no restriction at all with respect to external actions.

However, we have to adopt some special measures to avoid transactions that execute irrecoverable external actions to be undone at partition merge. A way to assure this is to "mark" those transactions as "permanent" in the

partition-log and so even if it is involved in a cycle and is the transaction with less descendants it will not be chosen to be rolled-back. Another precaution should be taken to avoid that at partition merge irrecoverable external actions be executed again. This is achieved directly by the proposed approach because transactions are not executed in the other partition, but only the values of the modified objects are forwarded.

V. CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH

In this thesis we analyzed the problem presented by network partitions on distributed database systems, with replicated data. Because of the need to preserve mutual consistency and to avoid irrecoverable external actions which may be performed by transactions operating on inconsistent data, it is impossible to allow unrestricted operation under network partitions. Consistency and availavility just appear to be fundamentally incompatible goals.

Existing solutions allowing one operating partition totally block update transactions in all but one partition. In this way mutual consistency is guaranteed when databases from different partitions are merged. Howevwer, these solutions are not acceptable for many existing military and commercial applications which require high availability. For example, an airline reservation system will prefer to conditionally reserve a seat on a flight for a customer rather than telling him to wait until the partition is repaired.

The two recently proposed solutions which allow multiple operating partitions greatly increase availability of distributed systems allowing them to utilize more fully the potential improvement provided by redundant data. The method

consisting of the version vector mechanism together with the log filter is very simple in structure and involves the addition of only a few new constructs to the file system design, namely, file origin points, version vectors, and the log filter. Thus this approach requires very little system overhead. However, the approach is specially suited for an environment in which file updates are moderate and conflicts occur only rarely and thus it will probably be not very useful in the kind of environment characterizing a database system with high transaction rates and volatility.

The approach involving semantic knowledge is based on the addition of semantic constraints to those already existing within a particular application. These constraints are enforced by the DBMS through the use of strong data types and integrity assertions. The use of semantic information about data assures that conflict detection and database reconciliation can be performed when the network is reconnected. This is perhaps the best existing solution to the network partition problem since it allows the highest degree of availability through the use of different classes of semantics of operations. Therefore, in most of the cases the database can be reconciled without the necessity of rolling-back transactions which had been executed in partitioned mode in order to achieve mutual consistency and thus the user will feel confident that even in the event of a network partition his transactions are going to be executed

giving reliable results. However, this approach lacks generality since the use of data types restricts access to the database to a set of limited operations. Also for each different application semantic information about the operations used in each them must be given to the DBMS in order to correctly reconcile the database. Clearly the overhead incurred by the reconciliation algorithms and the extra information required will increase proportionally with the number of applications processed by a system.

The approach proposed in this thesis assumes no semantic knowledge about the data and thus is more general since no predefined operations are required and several applications will not increase the amount of information necessary for the reconciliaton algorithm. It can be argued that a serious disadvantage of this method is that the way in which mutual consistency is achieved is by rolling-back conflicting transactions and then reexecuting them again and thus final results may be different from the ones obtained by the users when the network was partitioned. However, rolling-back transactions should not be the rule but the exception since in a large class of applications most transactions will never interfere with each other [5],[6]. Also in the uncommon case in which a conflict is detected the reconciliation algorithm will roll-back transactions only as a last resource when copies in different partitions of the same logical data-object had been independently updated. Even in

this case it is attempted to minimize the number of transactions that need to be rolled-back in one partition by choosing the transaction with less descendents.

Further research is needed in this area in order to determine which is the best method for dealing with network partitions for different applications. Perhaps a combination of semantic knowledge with the approach presented in the thesis will be the most appropriate for some applications. For example, in many commercial and military applications class A semantics is the most frequent [8] and since it is the class of semantics with less associated overhead, it can be used together with the alternate approach presented here in order to reduce the amount of semantic information required by the DBMS and thus reducing the overhead.

LIST OF REFERENCES

1. Alsberg, P.A. & Day, "A Principle for Resilient Sharing of Distributed Resources", Proceedings of the 2nd International Conference on Software Engineering 13-15 October 1976.

2. Bayer R., "On the Integrity of Database and Resource Locking". Lecture Notes in Computer Science: Data Base Systems, edited by Goos G. and Hartmanis J., Springer-Verlag, 1975, pp. 339-361.

3. Badal, D.Z., and Popek, G.J. "A Proposal for Distributed Concurrency Control for Partially Replicated Distributed Databases", Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, August 1978.

4. Badal, D.Z., "Concurrency Control Overhead or Closer Look at Blocking vs. Nonblocking Concurrency Control Mechanisms", Proceedings of the Fifth Berkeley Workshop on Distributed Data Management and Computer Networks, San Francisco, February 1981.

5. Bernstein, P.A., Shipman D.W., Rothnie J. B., "Concurrency Control in a System for Distributed Database (SDD-1)", ACM Transactions on Database Systems, March 1980, pp. 18-51.

6. Bernstein P.A., Shipman D.W. "The Correctness of Concurrency Control Mechanisms in a System for Distributed Databases (SDD-1)", ACM Transactions on Database Systems, March 1980, pp. 52-68.

7. Eswaran, K.P., J.N. Gray, R.A. Lorie, I.L. Traiger, "The Notions of Consistency and Predicate Locking in a Database System", Communications of the ACM, Vol 19, no. 11, November, 1976.

8. Faissol, S.Z., Operation of Distributed Database Systems Under Network Partitions, Ph.D. Thesis Dissertation, Computer Science Department, University of California, Los Angeles, July 1981.

9. Gifford, D.K., "Weighted Voting for Replicated Data", 7th Symposium on Operating Systems Principles, December 1979, pp. 150-162.

10. Gray, J.N., R.A. Lorie, G.R. Putzulu, I.L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base", Modelling in Data Base Management Systems, G.M. Nijssen (editors), North Holland Publishing Company, 1976.

11. Gray, J.N., "Notes on Data Base Operating Systems", Operating Systems: an Advanced Course, Springer-Verlag, Berlin, 1978.

12. Hammer, M. & D. Shipman, "An Overview of Reliability Mechanisms for a Distributed Data Base System", Spring COMPCON 78, San Francisco, February 1978, pp.63-65.

13. Horowitz, E.and Sahni, S., Fundamentals of Data Structures, Computer Science Press Inc., 1976.

14. Kung, H.T., J.R. Robinson, "On Optimistic Methods for Concurrency Control", ACM Transactions on Database Systems, June 1981, pp. 213-226.

15. Menasce, D.A., G.J. Popek, R.R. Muntz, "A Locking Protocol for Resource Coordination in Distributed Systems", ACM Transactions on Database Systems, June 1980, pp. 103-108.

16. Parker, D.s., Popek, et. al., "Detection of Mutual Inconsistency in Distributed Systems", Proceedings of the Fifth Berkeley Workshop on Distributed Data Management and Computer Networks, San Francisco, February 1981, pp. 172-184.

17. Parker, D.S., R.A. Ramos, "A Distributed File System Architecture Supporting High Availability", Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks, Asilomar, February 16-19, 1982, pp. 161-183.

18. Thomas, R.H., " A Solution to the Concurrency Control Problem for Multiple Copy Databases", Spring COMPCON 78, San Francisco, February 1978, pp. 56-62.

19. Rothnie, J.B. & N. Goodman, "A Survey of Research and Development in Distributed Database Management", Proceedings of the Third Conference on Very Large Databases, Tokyo, October 1977, pp. 48-61.

20. Ullman, J.D., Principles of Database Systems, Computer Science Press Inc., 1980.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Defense Logistics Studies Information Exchange U.S. Army Logistics Management Center Fort Lee, Virginia 23801	2
3.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
4.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
5.	Department Chairman, Code 54 Department of Administrative Sciences Naval Postgraduate School Monterey, California 93940	1
6.	Dr. Norman R. Lyons, Code 54LB Department of Administrative Sciences Naval Postgraduate School Monterey, California 93940	2
7.	Dr. Dusan D. Badal, Code 52ZD Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
8.	Dr. Norman F. Schneidewind, Code 54SS Department of Administrative Sciences Naval Postgraduate School Monterey, California 93940	1
9.	Captain Peter L. Jones Marine Corps Central Design and Programming Activity Marine Corps Development and Education Center Quantico, Virginia 22134	1

10. LT. Ricardo Arana C. 1
Ministerio de Marina
Central de Procesamiento de Datos
Av. Salaverry s/n
Lima-Peru

11. LT. Javier De La Cuba B. 1
Ministerio de Marina
Direccion de Abastecimiento Naval
Base Naval del Callao
Lima-Peru

12. LT. Eduardo Bresani T. 2
Ministerio de Marina
Central de Procesamiento de Datos
Av. Salaverry s/n
Lima-Peru

13. LTJG. Suha Futaci 1
Ataturk Caddesi Petek
Apartamani Kat 3 Daire 9
Bursa Turkey